Refactoring and Expanding MATLAB MarsIce Modeling Software

Daniel Johnson, Advised by Hanna Sizemore

Document Management Information

- Revision Number: 1.0
- Revision Release Date: 2020-8-23
- Purpose of Revision: Final draft.
- Scope of Revision: All sections of document.

Purpose and Scope

This document contains a brief description of the `MarsIce` software and the modifications made to it during the summer of 2020. It also provides instructions on the use of the software's key features.

How This Document Is Organized

This document contains three primary sections: Architecture Background, Views and Reference Materials. The Architecture Background section contains the following subsections to provide a detailed description of the software and explanations of the changes made to the system. The Problem Background subsection describes the original state of the `MarsIce` software and the problems relating to its rigidness. The Goals and Context subsection details the objectives that were planned for completion during the summer of 2020 and the reasoning behind them. The Solution Background describes the current state of the `MarsIce` software and the changes we made to the system to meet the goals outlined in the previous section. The Requirements Coverage highlights the primary stakeholders and their relevant use cases with the corresponding View.

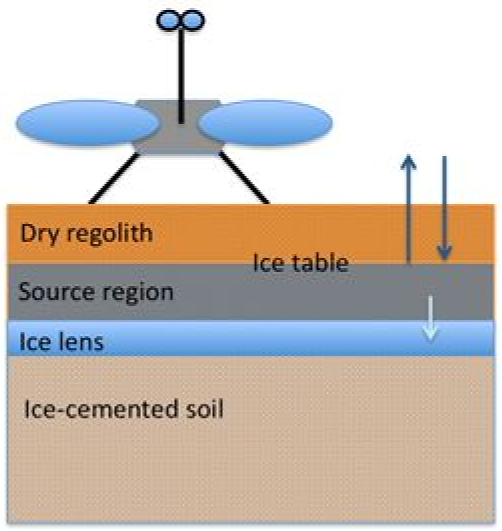
The Views section contains visual representations of the current `MarsIce` software which outline the current layout of the system.

The User Guide section goes into the specifics on how users can utilize the functionality of the current `MarsIce` software to generate data using existing soil types in addition to adding new soil types or modifying the properties of an existing type.

The References section contains citations of the resources used in this project.

Architecture Background

Problem Background



Schematic of the Martian subsurface in the vicinity of a growing ice lens. Dark blue arrows indicate the exchange of H_2O molecules between ground ice and the atmosphere via vapor diffusion. Light blue arrow indicates the migration of water to the ice lens via premelted films in the source region (Sizemore et al., 2016).

The original `MarsIce` software was developed in 2010 (Sizemore et al., 2015) using MATLAB to model the freezing behavior of water in a variety of soil types in Martian conditions. The software generated an assortment of output data files and figures, including the burial depth of ice with respect to changing Mars orbit parameters, ice temperature, liquid water fraction, and the heave rate or ice velocity over time.

The software was designed to aid understanding of how massive ice forms on Mars, motivated by Mars Odyssey Neutron Spectrometer and Phoenix lander observations indicating that subsurface ice volume was greater than expected for ice originating from vapor diffusion. `MarsIce` was designed to be run in concert with existing Mars climate models to search for conditions under which ice lenses could form and to estimate the rate of ice lens growth. Results from `MarsIce` can be used to determine if frost heave commonly occurs on Mars, the volume of clean ice produced at a given location, and whether other phenomena are more likely responsible for observed excess ice. The software package also has applications in habitability studies.

However, there were significant design problems with the original `MarsIce` software. Some of the software outputs were left over from debugging and were no longer needed. In addition, the software itself was very rigid, requiring changing a large portion of the source code in order to modify a soil type or salt doping. Recent significant progress in laboratory studies of Martian soil freezing and deliquescent salts have necessitated updates to the model. In order to expand upon the functionality of the `MarsIce` software, we had to refactor the existing codebase.

Goals and Context

The primary objectives during the summer of 2020 was to refactor the existing codebase for future expansions and to expand the original code's functionality to include new soil types with variable salt doping. The code also lacked a formal testing framework.

Solution Background

Approaches

Our work done on the `MarsIce` software can be broken down into the following sections. The sections are ordered according to when each section was completed in the project. This section describes the approaches in detail, while the User Guide section contains the information for how to use the new features that were added.

Refactored legacy code to utilize Object-Oriented Design

Before modifying or adding to the functionality of the project, we had to refactor the software to reduce the complexity and risk of building onto the existing `MarsIce` software. Since MATLAB supports Object-Oriented Design, we were able to use Objects to clean up a large amount of code duplication. We also cleaned up the messiest code -- i.e., code that contained many "bad smells," (Fowler, 2018). For a deeper explanation of the various types of bad code smells, details are described in Martin Fowler's book *Refactoring: Improving the design of existing code.* We refactored until the software was in a state in which we could write automated regression tests to ensure consistency of outputs moving forward.

Automated testing

Once the software was in a testable state, we utilized MATLAB's unit testing functionality to create an automated testing harness to ensure the program would behave consistently after modifications. Once the tests were in place, we could make further modifications to the system without risking accidentally modifying the program's behavior.

Developing the tests highlighted a key issue with MATLAB's ode15s solver: when the same software version was run on different computers, the outputs of the solver were different. Our first attempt at solving this issue was to use an error tolerance in the tests' assertion statements, as the discrepancies at first glance looked small enough to be attributed to rounding errors. The main issue that arose from this approach is that the automated tests would be unable to detect a slight change in the `MarsIce` outputs as long as the change is within the tolerance. The whole purpose of the tests is to detect any variation in the outputs, so we searched for another approach.

We experimented with alternatives to the ode15s solver to see if another solver could provide consistent results on different computers, however ode15s was the only one to successfully complete its calculations within a reasonable time frame. All other MATLAB solvers either took much longer to run into the same problem of producing different results over different computers or caused MATLAB to freeze.

The approach that we ultimately adopted was to utilize the mocking feature in MATLAB to mock the solver. This is a viable approach because the tests are testing the `MarsIce` software, not the MATLAB solver code. To implement the mocking, we saved a standard set of the expected inputs and outputs. The tests check to ensure the solver is called with the specific inputs and then provides the specific solver outputs for the rest of the `MarsIce` program to use. This standardizes the discrepancy across different computers, as each would be expecting the same specific input and providing the same specific output.

Extensible soil types and centralized soil properties

In order to add additional soil types, we had to construct an appropriate framework. In the original `MarsIce` code, the soil properties were scattered across the software files, making modifying or adding new soils high risk and difficult. Utilizing Object-Oriented Design, we changed the code structure so each soil type became its own class with a `DirtType` superclass. We were able to completely remove a large amount of code duplication involving soil name if/else chains by making a single call to a `DirtType` method. In addition, we stored soil specific properties as fields of their respective class, centralizing the soil properties. New soil types could be added by creating a new `DirtType` subclass and implementing `DirtType`'s abstract methods and adding the option to choose the new class in `DirtType`'s `getType` method, which is responsible for returning the soil type the user specified.

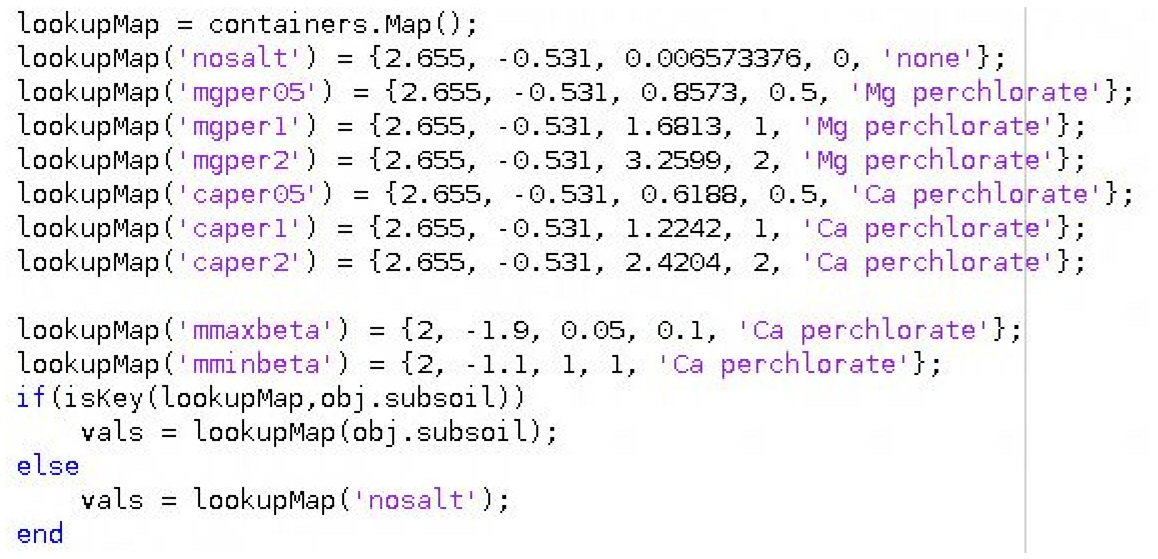
Standardized file naming

As we expanded the tests to cover more soil types, the amount of output files generated by `MarsIce` added excessive clutter to the root directory of the project. Since the soil properties were now centralized, we utilized the same structure to save output files. Each output file now is saved in the outputs folder instead of the root directory and all files have the same suffix structure containing the input parameters so the user knows what parameters created each file. We then refactored the automated tests to make use of the standardized names, resulting in the removal of a large amount of unnecessary code duplication.

Two new soil types (mminbeta and mmaxbeta)

Once we created the framework for adding new soils, the addition of two soil types was now possible. Our first approach was to create a new soil type class for each soil, however we discovered it would add excessive code bloat as these new types were variations of the Phoenix soil type. It made more sense to make soil types able to take a subsoil name parameter, with the side benefit of making the soil type identifiers more human readable. Now, the `DirtPhoenix` class contains a map object that maps the subsoil name to the specific parameters of that subsoil.

Another upgrade we completed was removing `dtf` as an input parameter, as the subsoil parameter replaced its functionality while making the code more readable. The `dtf` parameter previously determined the specific parameters of the Phoenix soil, which is no longer needed with the `subsoil` parameter. An example of one of the uses of the subsoil parameter is shown below. The supported `subsoil` values are mapped to their specific properties. Based on the value of `subsoil` imputed to the program, the corresponding properties are retrieved.



```
lookupMap = containers.Map();
lookupMap('nosalt') = {2.655, -0.531, 0.006573376, 0, 'none'};
lookupMap('mgper05') = {2.655, -0.531, 0.8573, 0.5, 'Mg perchlorate'};
lookupMap('mgper1') = {2.655, -0.531, 1.6813, 1, 'Mg perchlorate'};
lookupMap('mgper2') = {2.655, -0.531, 3.2599, 2, 'Mg perchlorate'};
lookupMap('caper05') = {2.655, -0.531, 0.6188, 0.5, 'Ca perchlorate'};
lookupMap('caper1') = {2.655, -0.531, 1.2242, 1, 'Ca perchlorate'};
lookupMap('caper2') = {2.655, -0.531, 2.4204, 2, 'Ca perchlorate'};

lookupMap('mmaxbeta') = {2, -1.9, 0.05, 0.1, 'Ca perchlorate'};
lookupMap('mminbeta') = {2, -1.1, 1, 1, 'Ca perchlorate'};
if(isKey(lookupMap,obj.subsoil))
    vals = lookupMap(obj.subsoil);
else
    vals = lookupMap('nosalt');
end
```

In addition, a large amount of if/else statements based on `dtf` were replaced with accessing a map using `subsoil` as the key, as shown above. The current parameters used in each `DirtType` subclass are shown in the Class Diagram in the Views section.

Improved viscosity calculation

Note that the viscosity calculation improvement is still a work in progress. The original viscosity calculation function `etaval` assumed that ice formed from pure water. However, since the software can now support salty solutions, we began to upgrade the viscosity calculations. We added the function `conc` to calculate the salt concentration in the liquid water. The concentration is passed to the upgraded `etaval` to solve for the viscosity of the specified salt solution. The type of salt is accessed from `DirtType` using its `getSaltType()` method. However, the new equation we added to `etaval` to handle the salty solutions produced questionable outputs for one salt species. This equation was provided by an outside collaborator, Aaron Zent, based on fits to new laboratory data. Zent is currently investigating alternative functions to better fit the raw data. We decided to revert back to using the pure water model until this issue can be resolved.

The addition of the `conc` function and the structural changes to `etaval` that allow it to access information on salt via `DirtType` provide a framework that can be applied to the calculation of other parameters, such as permeability, as new laboratory data becomes available in the future.

Requirements Coverage

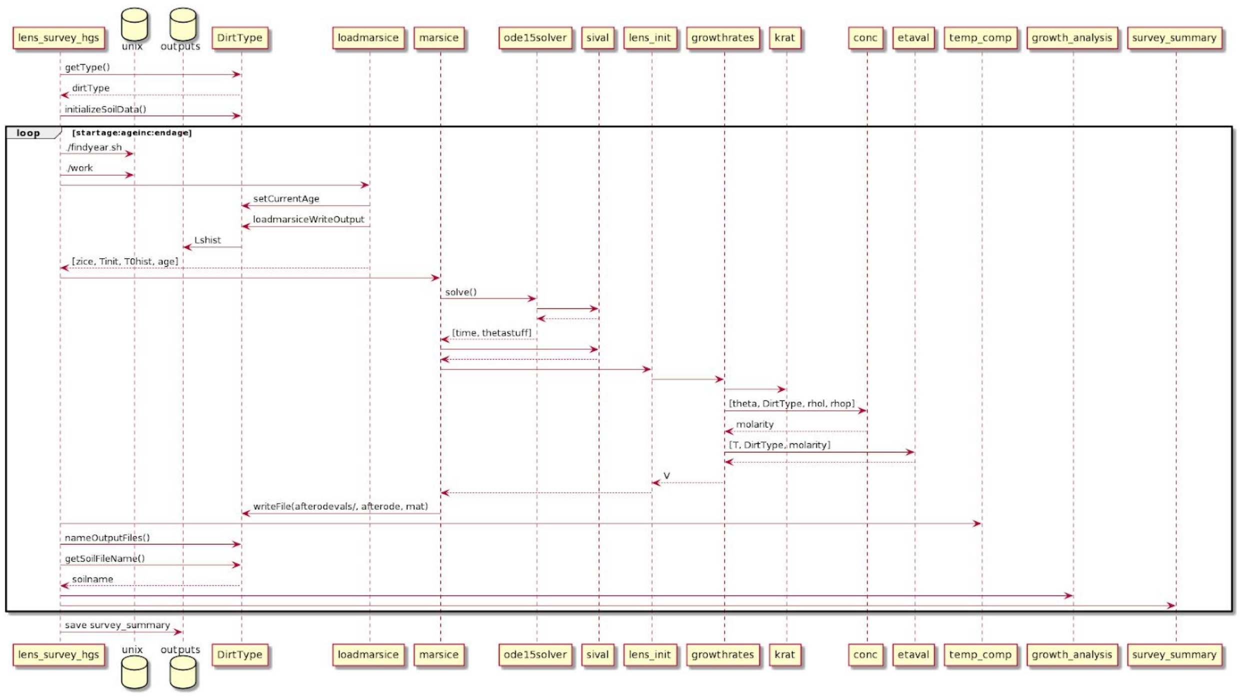
The primary stakeholders of the project are Developers and Scientists. Developers would be those who would be expanding or modifying the `MarsIce` functionality and the Scientists would be those running `MarsIce` with various input parameters but not modifying the source code.

Stakeholder(s)	Use Case	Related View
Developer and Scientist	Run lens_survey_hgs	Project File Structure
Developer	Run automated tests	Sequence Diagram
Developer	Edit a property of a soil	Class Diagram
Developer	Add new subsoils	Class Diagram
Developer	Add new soil type	Class Diagram
Developer	Add new test	Project File Structure
Developer	Modify source code	Sequence Diagram

Views

Sequence Diagram

Primary Presentation



Element Catalog

- `lens_survey_hgs` - the "Main" of the program
- `Unix` - A unix command being sent in MATLAB
- `Outputs` - the outputs folder in the Project File Structure
- `DirtType` - the superclass for soil types
- `loadMarsIce` - loads temperature and depth data generated from the `work` executable
- `MarsIce` - primary data processing function.
- `ode15solver` - uses `ode15s` to solve a partial differential equation for calculating temperature. When `lens_survey_hgs` is run from the testing framework, a mocked instance of `ode15solver` is passed in.
- `sival` - returns ice saturation.
- `lens_init` - extended data processing function
- `growthrates` - calculates the rate of growth of ice in the specified soil
- `krat` - returns permeability ratio for viscosity calculations
- `conc` - returns the concentration of salt concentration for viscosity calculations
- `etaval` - handles viscosity calculations
- `temp_comp` - compares the calculated depth of ice with respect to temperature over time to analytic models (Zent 2008, Mellon et al. 2004)

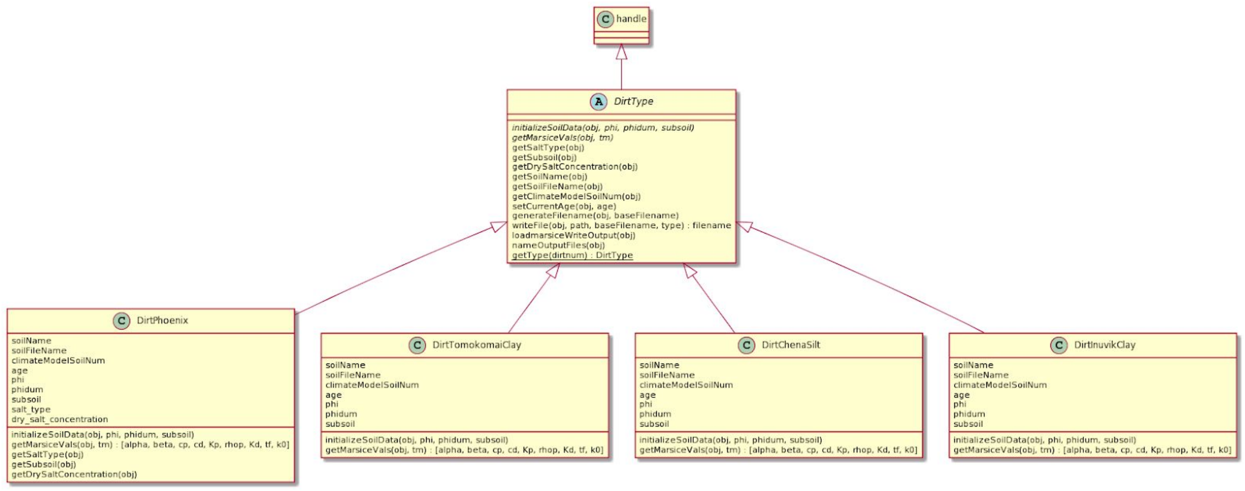
- `growth_analysis` - plots the results of `growthrates`
- `survey_summary` - collection of data calculated for a specific `age`

Design Background

The sequence diagram shows the major calls between the software files in the `MarsIce` project when lens_survey_hgs is called. The overall sequence of method calls is mostly preserved from the original `MarsIce` code. The major change to the call sequence was the addition of `DirtType`. Most of the code duplication and large if/else chains in the original software were replaced by method calls to `DirtType` shown in the diagram.

Class Diagram

Primary Presentation



C handle
A DirtType
initializeSoilData(obj, phi, phidum, subsoil)
getMarsiceVals(obj, tm)
getSaltType(obj)
getSubsoil(obj)
getDrySaltConcentration(obj)
getSoilName(obj)
getSoilFileName(obj)
getClimateModelSoilNum(obj)
setCurrentAge(obj, age)
generateFilename(obj, baseFilename)
writeFile(obj, path, baseFilename, type) : filename
loadmarsiceWriteOutput(obj)
nameOutputFiles(obj)
getType(dirtnum) : DirtType
C DirtPhoenix
soilName
soilFileName
climateModelSoilNum
age
phi
phidum
subsoil
salt_type
dry_salt_concentration
initializeSoilData(obj, phi, phidum, subsoil)
getMarsiceVals(obj, tm) : [alpha, beta, cp, cd, Kp, rhop, Kd, tf, k0]
getSaltType(obj)
getSubsoil(obj)
getDrySaltConcentration(obj)
C DirtTomokomaiClay
soilName
soilFileName
climateModelSoilNum
age
phi
phidum
subsoil
initializeSoilData(obj, phi, phidum, subsoil)
getMarsiceVals(obj, tm) : [alpha, beta, cp, cd, Kp, rhop, Kd, tf, k0]
C DirtChenaSilt
soilName
soilFileName
climateModelSoilNum
age
phi
phidum
subsoil
initializeSoilData(obj, phi, phidum, subsoil)
getMarsiceVals(obj, tm) : [alpha, beta, cp, cd, Kp, rhop, Kd, tf, k0]
C DirtInuvikClay
soilName
soilFileName
climateModelSoilNum
age
phi
phidum
subsoil
initializeSoilData(obj, phi, phidum, subsoil)
getMarsiceVals(obj, tm) : [alpha, beta, cp, cd, Kp, rhop, Kd, tf, k0]

Element Catalog

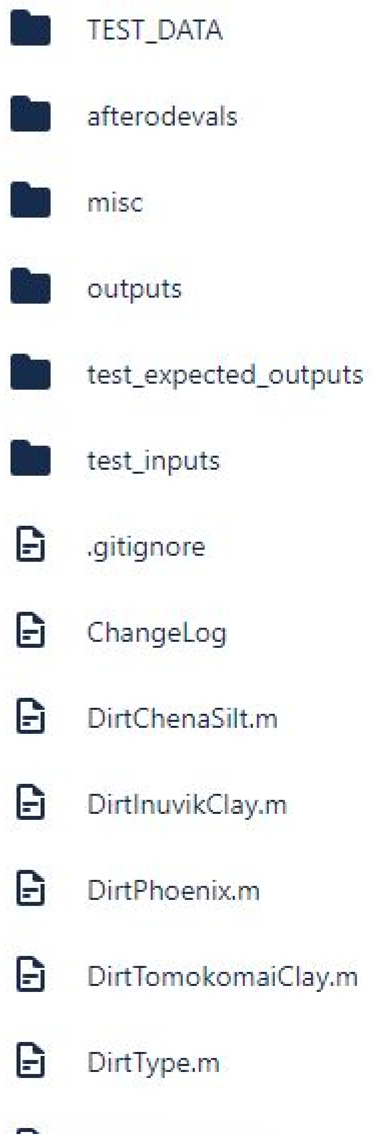
- `Handle` - base MATLAB object
- `DirtType` - soil type abstract superclass. All soil type classes extend this class and implement its abstract methods. `DirtType` cannot be instantiated since it is an abstract class
- `DirtPhoenix` - soil type class that models Martian soils at the Phoenix landing site. Currently the only soil type class that uses the `subsoil` parameter in its methods.
- `DirtTomokomaiClay` - soil type class that models Tomokomai clay.
- `DirtChenaSilt` - soil type class that models Chena silt.
- `DirtInuvikClay` - soil type class that models Inuvik clay.
- Italicized lines - abstract methods that any subclass must implement
- Underlined lines - static methods that can be called anywhere without an instance of the class

Design Background

The diagram serves to show the centralized soil type properties and what creating a new soil type class would require. In order to create new soil types, a new class would have to be created that extends `DirtType`. In order to be created correctly, the new class would have to implement `DirtType`'s abstract methods and contain the properties that the existing subclasses share. Note that a future improvement would be to move the shared properties into `DirtType`, however this will involve working with MATLAB's obscure functionality in dealing with accessing superclass properties from subclasses.

Project File Structure

Primary Presentation



TEST_DATA
afterodevals
misc
outputs
test_expected_outputs
test_inputs
.gitignore
ChangeLog
DirtChenaSilt.m
DirtInuvikClay.m
DirtPhoenix.m
DirtTomokomaiClay.m
DirtType.m

Element Catalog

- `afteroevals` - contains .mat files containing the values of all variables after the `ode15s` solver is called. These files can be used to set the standard values the mocked solver uses in the automated tests
- `misc` - contains diagram images and plantuml source code
- `outputs` - contains all files outputted by `MarsIce` with the input parameters appended to their name for easy recognition
- `test_expected_outputs` - contains output files that the automated tests expect `MarsIce` to output
- `test_inputs` - contains input files, one for each automated test
- `inputMarsIce.txt` (not shown) - the default file `lens_survey_hgs` loads input parameters from
- `lens_survey_hgs.m` (not shown) - the “Main” of the program
- `temperatureTests.m` (not shown) - run this file to run all the automated tests

Design Background

The structure of how the project directory is organized is used to help locate specific files.

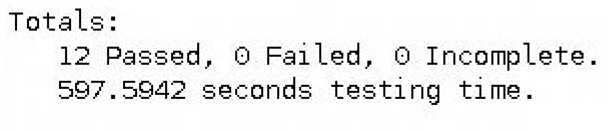
User Guide

Running lens_survey_hgs

The `lens_survey_hgs.m` file is what should be run to operate the `MarsIce` program. By default, it accepts no arguments, pulling input parameters from the `inputMarsIce.txt` file in the root directory. Other input files can be specified by calling `lens_survey_hgs.m` with two arguments. The first argument being the file path and the second being the file name. Lastly, a third argument can be supplied in the form of an `ode15solver`. This is used in the testing framework so the tests can supply the mocked solver to the program. All output files are saved in the `outputs` folder.

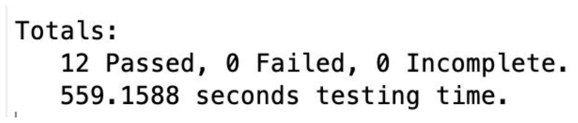
Running automated tests

Running the function `temperatureTests.m` runs all the automated tests. The tests require no arguments to run. Running all tests can be time consuming depending on the computer. The outputs of the tests should look similar to the test outputs shown below, which were generated on Maxwell, a linux server at GBO with 2.4 GHz Intel Xeon processors:



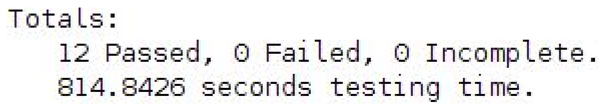
```
Totals:
   12 Passed, 0 Failed, 0 Incomplete.
   597.5942 seconds testing time.
```

Complete testing consists of running twelve individual 5-sol (Mars-day) simulations. All tests run for current Mars climate conditions with Ls = 145° and latitude = 70° N (specified in input files stored in the `test_inputs` directory). Testing time on a MacBook Pro with 2.9 GHz Intel Core i7 processor was slightly faster than on Maxwell:



```
Totals:
   12 Passed, 0 Failed, 0 Incomplete.
   559.1588 seconds testing time.
```

Running the same set of tests on a Linux virtual machine with a 2.9 GHz Intel Core i7-7820HQ processor is significantly slower:



```
Totals:
   12 Passed, 0 Failed, 0 Incomplete.
   814.8426 seconds testing time.
```

Editing soil properties

Each soil property is stored in the respective soil type class, e.g. `DirtPhoenix`. Some of the properties defined in the soil type classes are given default values, however they are updated to reflect the values of the input parameters if provided.

Adding new subsoils

Adding a new `subsoil` value requires adding a map if the soil type does not contain one already. The map uses the `subsoil` name as the key to map to its respective subsoil properties. Extending a map to contain a new subsoil only requires the addition of a new line of code providing the properties of the new subsoil.

Adding new soil types

Adding a new soil type requires the creation of a new MATLAB file. The new file must consist of a class that extends `DirtType` and implements the methods specified as abstract in `DirtType.m`, as well as contain the same properties as the other soil type classes (see the properties section at the top of `DirtPhoenix.m` for an example of the required properties). Lastly, the new soil type class must be added to `DirtType`'s `getType` method by assigning it a `soilnum` that isn't already used in the method.

Adding new tests

The testing framework has been condensed to run every test for every input. To run all tests for a new set of input parameters, add a new test method following the format of the existing tests by specifying the input parameters and calling the test frame method. In addition, standard output files and a standard `afterode` mat file must be generated by running `lens_survey_hgs` with the input parameters first, then copied into the `test_expected_outputs` folder. The input file used must be copied into the `test_inputs` folder and named using the same format as the existing input test files. When `temperatureTests.m` is run, the new test should execute along with the existing tests. Note that tests can be commented out to prevent them from running.

References

Sizemore, H.G., A. P. Zent, & A. W. Rempel (2015). Initiation and growth of martian ice lenses. Icarus, 251, 191-210. http://dx.doi.org/10.1016/j.icarus.2014.04.013

Sizemore, H. G. (2016). Frost Heave on Mars: Rate Limiting Boundary Conditions and the Role of Salts. Proposal submitted to NASA ROSES solicitation NNH16ZDA001N-SSW; award number 80NSSC18K0012.

Paul Williamson (2020). Strategy Design Pattern in MATLAB 2008b (https://www.mathworks.com/MATLABcentral/fileexchange/22193-strategy-design-pattern-in-MATLAB-2008b), MATLAB Central File Exchange. Retrieved July 20, 2020.

(n.d.). Testing Frameworks (https://www.mathworks.com/help/MATLAB/MATLAB-unit-test-framework.html?s_tid=CRUX_lftnav), Mathworks Help Center. Retrieved July 20, 2020

Fowler, M. (2018). Chapter 3. In *Refactoring: Improving the design of existing code*. Addison-Wesley.